SCHOOL OF COMPUTING

Title: Selective and recurring re-computation of Big Data analytics tasks: insights from a Genomics case study

Names: Jacek Cała, Paolo Missier

TECHNICAL REPORT SERIES

No. CS-TR- 1515 October 2017

TECHNICAL REPORT SERIES

No. CS-TR- 1515 Date 31/Oct/2017

Title

Selective and recurring re-computation of Big Data analytics tasks: insights from a Genomics case study

Authors

Jacek Cała, Paolo Missier

Abstract

In Data Science, knowledge generated by a resource-intensive analytics process is a valuable asset. Such value, however, tends to decay over time as a consequence of the evolution of any of the elements the process depends on: external data sources, libraries, and system dependencies. It is therefore important to be able to (i) detect changes that may partially or completely invalidate prior outcomes, (ii) determine the impact that those changes will have on those prior outcomes, ideally without having to perform expensive re-computations, and (iii) optimise the process re-execution needed to selectively refresh affected outcomes. This paper presents an extensive experimental study on how the selective re-computation problem manifests itself in a relevant analytics task for Genomics, namely variant calling and clinical interpretation, and how the problem can be addressed using a combination of approaches. Starting from this experience, we then offer a blueprint for a generic re-computation meta-process that makes use of process history metadata to make informed decisions about selective re-computations in reaction to a variety of changes in the data.

Printed and published by Newcastle University,
School of Computing, Urban Sciences Building, Science Central,
Newcastle upon Tyne, NE4 5TG, England.

Bibliographical details

Title and Authors
Jacek Cala, Paolo Missier

Selective and recurring re-computation of Big Data analytics tasks: insights from a Genomics case study
Jacek Cała, Paolo Missier

NEWCASTLE UNIVERSITY
School of Computing. Technical Report Series. CS-TR- 1515

Abstract
In Data Science, knowledge generated by a resource-intensive analytics process is a valuable asset. Such value, however, tends to decay over time as a consequence of the evolution of any of the elements the process depends on: external data sources, libraries, and system dependencies. It is therefore important to be able to (i) detect changes that may partially or completely invalidate prior outcomes, (ii) determine the impact that those changes will have on those prior outcomes, ideally without having to perform expensive re-computations, and (iii) optimise the process re-execution needed to selectively refresh affected outcomes. This paper presents an extensive experimental study on how the selective re-computation problem manifests itself in a relevant analytics task for Genomics, namely variant calling and clinical interpretation, and how the problem can be addressed using a combination of approaches. Starting from this experience, we then offer a blueprint for a generic re-computation meta-process that makes use of process history metadata to make informed decisions about selective re-computations in reaction to a variety of changes in the data.

About the authors

Dr Jacek Cała is a Senior Research Associate in the School of Computing Science at Newcastle University. He works on high performance cloud-based systems and their application to e-Science. His main interests include cloud computing, software deployment, component-based and distributed solutions driving workflows and e-Science. Previously, Jacek worked as a Teaching and Research Assistant at AGH-University of Science and

Technology in Kraków, Poland where he was one of the architects and key developers of TeleDICOM, a system which supports medical teleconsultations in over 20 hospitals and medical centres in the South of Poland.

Dr. Paolo Missier is a Reader (Associate Professor) in Large-scale Information Management with the School of Computing Science, Newcastle University, UK. His current research interests are in large-scale metadata analytics, that is, on novel applications of data analytics techniques to large corpora of metadata, and data provenance management and analysis in particular. He is the PI of a 3-year project grant from EPSRC (UK) for the “ReComp” project. Paolo holds a Ph.D. in Computer Science from the University of Manchester, UK (2007), a M.Sc. in Computer Science from University of Houston, Tx., USA (1993) and a B.Sc. and M.Sc. in Computer Science from Universita' di Udine, Italy (1990).

Suggested keywords

RE-COMPUTATION, BIG DATA ANALYSIS, KNOWLEDGE DECAY, GENOMICS

Selective and recurring re-computation of Big Data analytics tasks: insights from a Genomics case study

Jacek Cała[a,*], Paolo Missier[a]

[a]School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU UK

Abstract

In Data Science, knowledge generated by a resource-intensive analytics process is a valuable asset. Such value, however, tends to decay over time as a consequence of the evolution of any of the elements the process depends on: external data sources, libraries, and system dependencies. It is therefore important to be able to (i) detect changes that may partially or completely invalidate prior outcomes, (ii) determine the impact that those changes will have on those prior outcomes, ideally without having to perform expensive re-computations, and (iii) optimise the process re-execution needed to selectively refresh affected outcomes. This paper presents an extensive experimental study on how the selective re-computation problem manifests itself in a relevant analytics task for Genomics, namely variant calling and clinical interpretation, and how the problem can be addressed using a combination of approaches. Starting from this experience, we then offer a blueprint for a generic re-computation meta-process that makes use of process history metadata to make informed decisions about selective re-computations in reaction to a variety of changes in the data.

Keywords: re-computation, big data analysis, knowledge decay, genomics

1. Introduction

A general problem in Data Science is that the knowledge outcomes generated through resource-intensive data analytics processes are subject to decay over time. This is often a consequence of the evolution of any of the elements the process depends on: external data sources, libraries, and system dependencies, as well as of the processes themselves. This paper is concerned with the problem of selectively refreshing some of those knowledge outcomes in reaction to any of the changes that may have affected their quality, by partially or entirely re-executing the corresponding processes. Any such change raises three questions: (i) which of the prior outcomes are affected by the change and are partially or completely invalidated, (ii) how much marginal benefit would be accrued by refreshing a particular outcome, and (iii) how much would it cost.

Two extreme but naive approaches, namely to re-compute every outcome in reaction to any change, and to do nothing, are equally unsatisfactory. The former, which we refer to as *continuous blind re-computation*, is likely to be inefficient as it can be expensive and possibly produce marginal improvements. On the other hand, ignoring changes altogether and continuing to use increasingly stale knowledge, while sometimes acceptable, is often risky. The genomics case study that underpins our experiments, introduced later in this section, is a prime example. Clinical diagnoses of genetic diseases are increasingly the result of processing genome data using Next Generation Sequencing (NGS) pipelines, at decreasing but still non-negligible cost per patient [1]. The genetics knowledge used by the process is encoded in a variety of genomics databases such as OMIM GeneMap,[1] NCBI ClinVar,[2] HGMD,[3] and many others.[4] As these continually evolve along with sequencing technology and software tools, opportunities arise to improve the accuracy of past diagnoses. Conversely, not reacting to such evolution risks leaving patients with inaccurate and uncertain information about their destiny. A quantitative assessment of the impact of these changes is provided in Sec. 1.2.

We are, thus, motivated to investigate techniques and strategies for deciding when and how to react to changes, while optimising the use of the available computing resources vis-à-vis the expected benefit of knowledge refresh on a popoulation of prior outcomes. We refer to this as the *selective re-computation* problem. In this paper we study several optimisations over the *blind re-computation* baseline, which are possible provided detailed records of past executions are available for analysis.

The work presented here is carried out as part of the ReComp project.[5] Although the project's ambition is to develop generic techniques that are applicable to a broad

*Corresponding author

Email addresses: Jacek.Cala@ncl.ac.uk (Jacek Cała), Paolo.Missier@ncl.ac.uk (Paolo Missier)

[1] http://data.omim.org
[2] https://www.ncbi.nlm.nih.gov/clinvar
[3] http://www.hgmd.cf.ac.uk
[4] http://grenada.lumc.nl/LSDBlist/lsdbs
[5] recomp.org.uk

range of data analytics processes, in this paper we focus on a singularly important and timely use case in genomics. It is a complex and resource-intensive process of identifiying potentially deleterious genomic mutations in humans, in order to help with the diagnosis of rare genetic diseases.

1.1. Reference case study: Genetic variants analysis

Over the last decade, Next Generation Sequencing technologies have made it possible to investigate the genetic mechanisms behind some of the most severe human diseases [2, 3, 4]. In this setting a typical variant calling pipeline takes an input genome, or its *exome*[6], and identifies all variants relative to the current reference human genome.[7] NGS pipelines are exemplary resource-intensive analytics processes: even when the analysis is performed on the exome, which includes only about 2% of the whole genome, the input files are of the order of 10GB each, and a batch of 20–40 exomes is required for the results to be significant. This 1TB+ input dataset requires over 100 CPU-hours to process. Specific performance figures for our own pipeline implementation, which runs on the Azure cloud, can be found in [1].

Once the variants (about 20,000) have been identified, a second process of *variant interpretation* selects the few, usually less than 100, that are most likely to provide insight to clinicians regarding a patient's suspected genetic disease, or *phenotype*. A number of freely available reference databases, compiled from the genetics literature, are used to address this "needle in the haystack" problem. One instance of such a process is Simple Variant Interpretation (SVI) [5], developed in our group, which uses knowledge from the ClinVar and Gene Map reference databases. Fig. 1 depicts the combination of Variant Calling and SVI.

In more detail, the SVI pipeline consists of three main steps (Fig. 2): (1) mapping the user-provided clinical terms that describe a patient's phenotype to a set of relevant genes (*genes-in-scope*), (2) selection of those variants that are in scope, that is, the subset of the patient's variants that are located on the genes-in-scope, and (3) annotation and classification of the variants-in-scope according to their expected pathogenicity. Classification consists of a simple traffic-light system {red, green, and amber} to denote pathogenic, benign and variants of unknown or uncertain pathogenicity, respectively. In this process, the class of a variant is determined simply by its *pathogenicity status* as reported in ClinVar. Importantly, if any of the patient variants is marked as red, the phenotype hypothesis is deemed to be confirmed, with more red variants interpreted as stronger confirmation.

The experiments carried out for this paper concern the SVI part of the pipeline alone (but will be extended to the full NGS pipeline in the future), for two reasons. Firstly, SVI is much less resource-intensive and thus easier to work with than the complete NGS processing pipeline, requiring a few minutes for each execution, and unlike the NGS pipeline, it is deterministic. At the same time, despite its small scale it is an exemplary use case for selective re-computation as both reference databases are updated frequently. This provides real examples of evolving data dependencies that may cause a patient's diagnosis to change over time, namely when new variants are added or when their *pathogenicity status* is updated. Indeed, the reliability of the diagnosis depends upon the content of those databases. While the presence of deleterious variants may sometimes provide conclusive evidence in support of the disease hypothesis, the diagnosis is often not conclusive due to missing information about the variants, or due to insufficient knowledge in those databases. As this knowledge evolves and these resources are updated, there are opportunities to revisit past inconclusive or potentially erroneous diagnoses, and thus to consider re-computation of the associated analysis. Furthermore, patient's variants, used as input to SVI, may also be updated as sequencing and variant calling technologies improve. The problem of deciding when variant selection and classification should be refreshed is therefore very concrete.

1.2. Extent of changes in the SVI case study

To clarify the need to perform selective re-computation on SVI, we conducted preliminary experiments to quantify the extent of changes in one of the two reference databases (ClinVar), and their consequences on patients' diagnoses. We analysed the variants for a cohort of 33 patients for three distinct phenotypes: *Alzheimer's disease*, *Frontotemporal Dementia-Amyotrophic Lateral Sclerosis* (FTD-ALS) and the *CADASIL syndrome*. For each patient we ran SVI 16 times, using consecutive monthly versions of ClinVar, from July 2015 to October 2016, and recorded whether the new version would have modified a diagnosis that had been obtained using the previous version. Recall that diagnosis is based on the classification of the few relevant variants. A change in diagnosis occurs when new variants are added to the selection, others are removed, or existing variants change their classification because their status in ClinVar has changed.

Table 1 summarises the results. We recorded four types of outcomes. Firstly, confirming the current diagnosis (■), which happens when additional variants are added to the `red` class. Secondly, retracting the diagnosis, which may happen (rarely) when all `red` variants are retracted, denoted ❖. Thirdly, changes in the `amber` class which do not alter the diagnosis (□), and finally, no change at all (·).

These results, however limited in their scope, confirm our assumption that strategies for selective re-computation of patients cases are needed. The majority of the changes reported here are ultimately of low interest to clinicians, and blind re-computation would be wasteful indeed. It comes as little surprise because some human genetic diseases tend to be underpinned by a very few rare vari-

[6] The exome consists of the areas within a gene that are transcribed and thus participate in protein synthesis.

[7] https://software.broadinstitute.org/gatk/best-practices

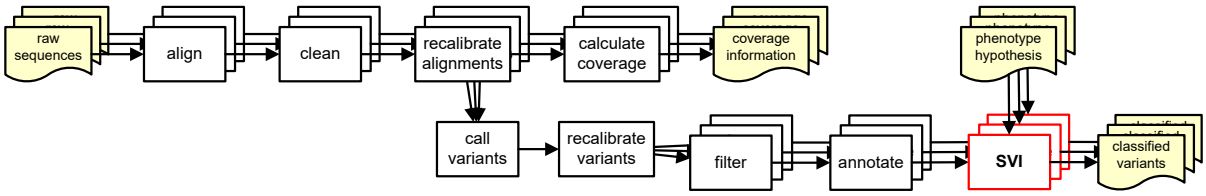

Figure 1: The Next Generation Sequencing pipeline; highlighted is the variant classification step.

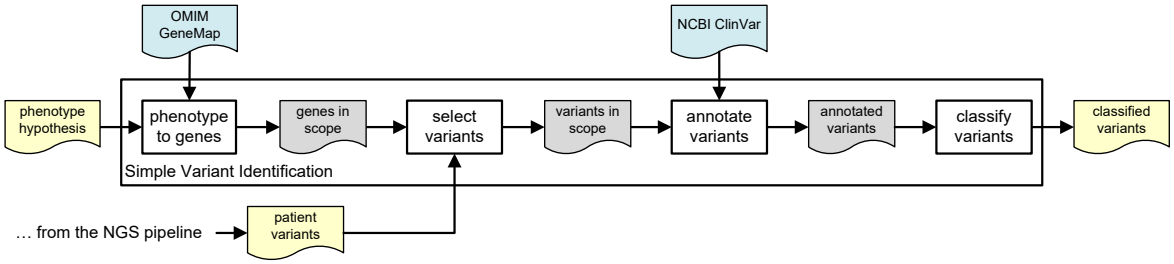

Figure 2: The high-level architecture of the SVI process.

Table 1: Changes observed in the output of the SVI tool for a cohort of 33 patients following updates in the NCBI ClinVar reference database between July 2015 and October 2016.

Phenotype hypothesis	Frontotemporal Dementia-Amyotrophic Lateral Sclerosis												CADASIL	Alzheimer's disease																	
Variant file / ClinVar version	D_1071	D_1049	D_1041	D_0899	D_0854	D_0830	C_0171	C_0098	C_0056	C_0053	C_0051	B_0307	D_1136	C_1457	C_0072	C_0071	C_0068	C_0065	B_0396	B_0384	B_0370	B_0365	B_0358	B_0338	B_0331	B_0229	B_0214	B_0209	B_0208	B_0203	B_0202	B_0201	B_0198
08/15	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
09/15	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
10/15	·	·	·	·	·	·	·	·	·	·	·	·	■	□	·	·	·	□	·	·	·	□	·	·	□	□	·	□	□	·	·	□	□
11/15	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
12/15	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
01/16	·	·	·	·	·	·	·	·	·	·	·	·	·	□	□	·	□	□	□	·	□	□	□	□	□	□	□	□	□	□	·	□	□
02/16	·	·	□	·	□	·	□	·	□	□	□	□	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
03/16	□	·	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
04/16	·	·	·	·	·	·	·	·	·	·	·	·	·	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
05/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	□	·
06/16	·	·	□	·	□	·	□	□	·	·	□	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
07/16	·	·	□	·	□	·	·	·	□	·	□	·	·	·	□	·	·	·	·	□	□	·	·	□	·	□	□	·	□	□	□	□	□
08/16	·	·	·	·	·	·	·	·	·	·	·	□	·	❖	■	❖	□	■	□	·	■	■	■	■	□	■	■	□	□	■	·	□	□
09/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	□	·	□	□	□	·	□	□	□	□	□	□	□	□	·	□	·	·	□
10/16	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□

ants [6], whilst those associated with common diseases (as above) are widely studied. Therefore, the knowledge about them is quite stable, especially when considered on a monthly time scale. This also suggests that rare diseases may provide a more compelling case for selective re-computation, as knowledge about them is more likely to evolve over time. Finally, we note that some updates have a higher impact than others, for instance August 2016.

1.3. Approaches to selective re-computation

This paper is focused on *lossless selective re-computation*, which seeks to reduce the amount of processing performed in reaction to each change, relative to blind re-computation, while still ensuring that each outcome on which the change has non-zero impact is indeed updated. In this setting, we address the selective re-computation problem in three complementary ways.

P1: Partial re-execution.

Firstly, if we can analyse the structure and semantics of process P, to recompute an instance of P more effectively we may be able to reduce re-computation to only those parts of the process that are actually involved in the processing of the changed data. For this, we are inspired by techniques for *smart rerun* of workflow-based applications [7, 8], as well as by more general approaches to incremental computation [9, 10].

The basic requirement to support partial re-execution is that intermediate data of every past execution be cached, so it can be re-used in lieu of re-executing part of a process that is known to have not been affected by the changes. For example, if the updated data is used only in the middle of the process we may be able to skip processing of its initial part and use previously computed intermediate data instead. This is much easier to achieve in the case of scientific workflows (dataflows) such as our NGS pipeline, where the data dependencies are explicit. In this scenario, the main difficulty is to find a good balance between how much intermediate data should be cached, versus how much we should re-generate in order to minimise the overall re-computation cost (monetary, runtime and/or storage). Some guidance on this topic has been presented e.g. in [11, 12]. We present our experiments in Sec. 3.

P2: Differential execution.

The second approach to optimising re-computation involves analysis of the differences betwen versions of input and reference data. That is especially important in the big-data analyses, when changes often involve only small parts of a large dataset. Whilst blind re-computation uses the complete new version of the input data, for structured or semi-structured data and specific processes it may be possible to calculate and then use the difference sets of `added`, `removed` and `updated` records between the new and old version. However, whether or not it makes sense to execute a process using a *difference set* as its input depends on the function it implements. While specific cases are covered by existing research on incremental computing, a general formulation of differential execution, such as self-adjusting computation [9], still requires more effective realisation in practice [13]. In Sec. 4 we present our experiments and observations involving re-computation of SVI using difference sets.

P3: Identifying the scope of change.

Finally, we explore re-computation that ranges over a whole population of outcomes from past executions. For instance, NGS pipelines are now routinely used to process large patient cohorts. Thus, when a change in the input or reference data occurs, a natural question arises: which of the past outcomes (patient diagnoses) are affected by the change? In the example presented in Tab. 1, a re-computation framework that could use an oracle to answer this question would fire SVI only 14 times: once for each ClinVar 08/15, 10/15, 11/15 and eleven times for ClinVar 08/16. We show experimentally (Sec. 5) how we can decrease the number of required re-computations from 495 down to 71. This, combined with the ability to run SVI selectively as in (P1), (P2) above, allows us to perform lossless re-computation below the 10% of the runtime of the blind approach for most of the change cases and below 40% for all the cases.

1.4. Paper Contributions

In the rest of the paper we present our approach to addressing each of the three problems just outlined, and report on experiments to show their viability on the SVI case study.

Our first contribution is a formalisation of a reference framework for selective re-computation, which we use to formulate problems P1–P3. **Our second contribution** is an extensive experimental study, conducted using the SVI process as testbed, to determine the viability of: partial execution (P1 – Sec. 3), differential execution (P2 – Sec. 4) and scope determination (P3 – Sec. 5); and to quantify their benefits and assess their limitations.

The lossless approach to selective re-computation is conservative, in that any outcome on which the impact cannot be proved to be zero, regardless of how small, will be refreshed. In contrast, *lossy selective re-computation* also seeks to reduce the amount of re-computation performed on previously computed outcome. However, in this case we try to quantitatively *estimate* the extent of the impact, and use the estimates to decide on whether and when to refresh the outcomes. We view this as a more general decision problem in the re-computation space, which involves cost/benefit analysis. **Our final contribution** is an outline of the challenges in addressing this problem, and ideas for a technical approach (Sec. 6).

2. Formalisation and experimental testbed

In this section we provide a simple reference framework for expressing problems P1–P3, and introduce the techni-

cal elements that underpin our experiments.

2.1. Reference framework for selective re-computation

Consider an instance P_i of a deterministic process P, which takes input x_i and produces output y_i, using reference datasets $D = \{D_1 \dots D_m\}$. D is typically the same set across inputs. For SVI, $D = \{OM, CV\}$ consists of the two reference databases, OMIM GeneMap and NCBI ClinVar as mentioned above. Each x_i and y_i are specific inputs and outputs, respectively. These may be tuple-valued: $x_i = \langle x_{i1} \dots x_{in} \rangle$, $y_i = \langle y_{i1} \dots y_{im} \rangle$. We denote the types of x_{ij} (resp y_{ij}) in each instance x_i (resp y_i) with X_{ij} (resp Y_{ij}).

For instance, in SVI, patient i is represented by input $x_i = \langle x_{i1}, x_{i2} \rangle$ where x_{i1} is the list of patients variants and x_{i2} is a set of phenotype terms. The patient's diagnosis is the single output $y_i = \langle y_{i1} \rangle = \{(v, c)\}$, a set of variants v along with their class label c. for simplicity of notation, and without loss of generality, in the following we are going to refer to inputs and outputs simply as x_i, y_i, and to their types as X, Y.

Data versions and change notation. We assume that each of the x_i and each $D_j \in D$ may have multiple versions, which change over time, and denote the version of x_i at time t as x_i^t, and the state of D_j at t as D_j^t.

We write $x_i^t \rightarrow x_i^{t'}$ to denote that a new version of x_i has become available at time t', replacing the version x_i^t that was current at t. Similarly, $D_j^t \rightarrow D_j^{t'}$ denotes a new release of D_j at time t'.

Executions. We denote the execution P_i of P that takes place at time t by:

$$\langle y_i^t, c_i^t \rangle = exec(P, x_i^t, D^t) \tag{1}$$

where $D^t = \{D_1^t \dots D_m^t\}$. c_i^t denotes the cost of the execution, for example a time or monetary expression that summarises the cost of cloud resources. We also assume for simplicity that P remains constant.

Current outcomes. Finally, by slight abuse of notation, with $Y^t = \{y_1^t, y_2^t \dots y_N^t\}$ we denote a set of N outcomes that are *current* at time t, i.e., each y_i^t is the latest in a series of values $y_i^{t_1} \dots y_i^{t_k}$ with $t_k \leq t$.

Diff functions. We assume one can define a family of type-specific *data diff* functions, which quantify the extent of changes that occur over time in either x, D_j, or y. Specifically:

$$diff_X(x_i^t, x_i^{t'}) \quad diff_Y(y_i^t, y_i^{t'}) \tag{2}$$

compute the differences between two versions of x_i of type X, and two versions of y_i of type Y. Similarly, for each source D_j,

$$diff_{D_j}(D_j^t, D_j^{t'}) \tag{3}$$

quantifies the differences between two versions of D_j. The values computed by each of these functions are type-specific data structures, and will also depend on how changes are made available. For instance, $D_j^t, D_j^{t'}$ may represent successive transactional updates to a relational database. More realistically in our analytics setting, and on a longer time frame, these will be two releases of D_j, which occur periodically. In both cases, $diff_{D_j}(D_j^t, D_j^{t'})$ will contain three sets of added, removed, or updated records. The only assumption we make on these functions is that they should all report a *Nil* difference when their inputs are identical: $diff_T(v, v) = Nil$ for any type T.

Impact of a change. We say that a change $D_j^t \rightarrow D_j^{t'}$ (resp. $x_i^t \rightarrow x_i^{t'}$) has *non-zero impact* on outcome y^t iff $diff_Y(y^t, y^{t'}) \neq Nil$, where $y^{t'}$ is the new outcome computed using $D_j^{t'}$ (resp. $x_i^{t'}$) in (1).

2.2. Selective re-computation problems

Using the framework introduced above, we formulate problems P1–P3 as follows. Firstly, note that *blind re-computation* following a change in a data dependency D_j: $D_j^t \rightarrow D_j^{t'}$ is simply the complete re-execution of (1) for each y^t in Y^t, by replacing D_j^t with $D_j^{t'}$.[8]

P1: Partial re-execution. Suppose P is defined as a workflow, described as a directed acyclic graph of k processing elements $P_1 \dots P_k$, connected through data dependencies. Given an execution of P as in (1) and changes of the form $D_j^t \rightarrow D_j^{t'}$ and/or $x^t \rightarrow x^{t'}$, we want to identify the minimal subset P' of $h \leq k$ processing elements in P, such that executing P' using $x^{t'}$ and $D_j^{t'}$ yields the same result as executing P entirely. Past research [7, 8] outlines conditions under which effective partial execution is viable, specifically when P has a dataflow structure.

P2: Differential execution. Given an execution of P as in (1) and changes as above, in some cases it may be possible to refresh an outcome y^t by re-computing P using only the *differences* between old and new versions of the inputs or of the data resources. As difference sets are much smaller than the entire inputs or reference data resources, especially in the case of big data problems, this may result in significant savings in computation time.

P3: Identifying scope of change. Suppose an outcome y^t is produced a D_j^t which is later updated: $D_j^t \rightarrow D_j^{t'}$. Intuitively, if $exec(P, x_i^t, D^t)$ has not used any of the data in $diff_{D_j}(D_j^t, D_j^{t'})$, then (as we have assumed that P is deterministic) $y^{t'} = exec(P, x_i^t, D^{t'}) = y^t$.

Thus, we may be able to determine with certainty, as required in our lossless re-computation setting, that some

[8] Note that if the change is $x_i^t \rightarrow x_i^{t'}$, then trivially only the executions on input x_i^t are performed.

of the outcomes y^t need not re-computing, provided we maintain a detailed account of exactly which data each execution of P has used, from each of its external data resources. Following this intuition, we address the problem to identify the elements from a population of outcomes Y^t that are *out of scope*, that is, those for which re-execution is certainly going to produce identical results given changes in any of its data dependencies.

2.3. Requirements and experimental setting

The architectural pattern we adopt is that of a meta-process (the `ReComp` process) that can provide at least two minimal capabilities relative to an underlying process P:

1. to monitor changes in the inputs, dependencies, and outputs of P, and to quantify them, e.g. by accepting type-specific $diff_X()$, $diff_D()$, and $diff_Y()$ functions, and
2. to control the partial or entire re-execution of P.

Underpinning these capabilities are a number of technical requirements regarding collecting and storing various kinds of metadata during execution, and performing analytics on it. Specifically:

1. **Transparency of the process structure.** Ideally, P should be a *white box* process, that is, it should be possible to inspect its internal structure to support partial and differential execution;
2. **Observability of process execution and provenance collection.** It must be possible to observe data production and consumption events that occur during execution, and use those to reconstruct the provenance of the outcomes;
3. **Cost monitoring.** Similarly, it must be possible to assess the detailed cost (execution time, storage volume) of each execution, as this is required to learn estimates of the future cost of re-execution;
4. **Reproducibility of P.** Finally, it must be possible to enact a new execution of P on demand.

Note that each of these requirements may be satisfied to different extents by different process specification and execution models. Not all computational models are friendly to our metadata analytics approach, either because provenance collection is available but not at a level of detail that is usable (the NoWorkflow system [14], for instance, is very good at monitoring any Python process but its provenance is too low-level to be used here), or is not provided at all. In particular, *black box* processes that do not reveal their internal structure, cost, or execution provenance, such as third party web services, are particularly hostile to our analysis.

Our experiments, however, are carried out on a platform that provides ideal support for these requirements: SVI is implemented as a workflow (described next), which provides both full transparency and fine-grained provenance collection and cost monitoring capabilities. Furthermore, the workflow is deployed on the Azure cloud, which provides an additional mapping of resources to price, through their cost model. Exploring the extent to which our results deteriorate as the requirements above are not met is currently out of our scope.

Finally, regarding Reproducibility (4), we note that actual re-computation of older processes P under slightly different conditions is not straightforward, as it may require redeploying P on a new infrastructure and ensuring that the system and software dependencies are maintained correctly, or that the results obtained using new versions of third party libraries remain valid. Addressing these architectural issues is a research area of growing interest [15, 16, 17], but not a completely solved problem.

2.4. The SVI workflow

SVI is the natural continuation of the more complex variant calling NGS pipeline, and is implemented using the same workflow technology, namely the e-Science Central (e-SC) platform [18], a cloud-based Workflow Management System designed for scientific data management and analysis. A screenshot of the SVI implementation in e-SC is shown in Fig. 3. To recall, e-SC supports a simple dataflow programming model where processing blocks are connected to each other using data links, in a DAG topology. Our performance analysis of the variant calling workflow is described in detail elsewhere [1].

The choice of using e-SC as a platform for our experiments satisfies all of the requirements listed above. Firstly, workflows are *white box* processes, where the partial re-execution problem translates into a problem of selecting a suitable sub-graph from the whole workflow DAG structure.

Secondly, e-SC automatically records the derivation history of every workflow output from the inputs, i.e. their provenance. The provenance traces are described using the ProvONE data model [19], which extends the standard PROV data model [20]. For the purpose of human-readable description, in this paper we use the PROV-N notation [21] to present relevant provenance fragments. This includes details required for re-execution, such as the parameter settings for the processing blocks, and the version of each library and data dependency at the time of execution.

Thirdly, detailed execution costs at the level of the single processing block, as well as data storage costs, are available either through e-SC or the underlying cloud deployment (Azure, in this instance). Finally, the dependency manager that oversees the execution of e-SC workflows provides the required levels of reproducibility, i.e. the ability to re-run old workflows on demand. This rich corpus of metadata enables the kind of analysis required by our techniques, for instance estimating the cost of selecting a particular sub-graph for partial workflow re-execution.

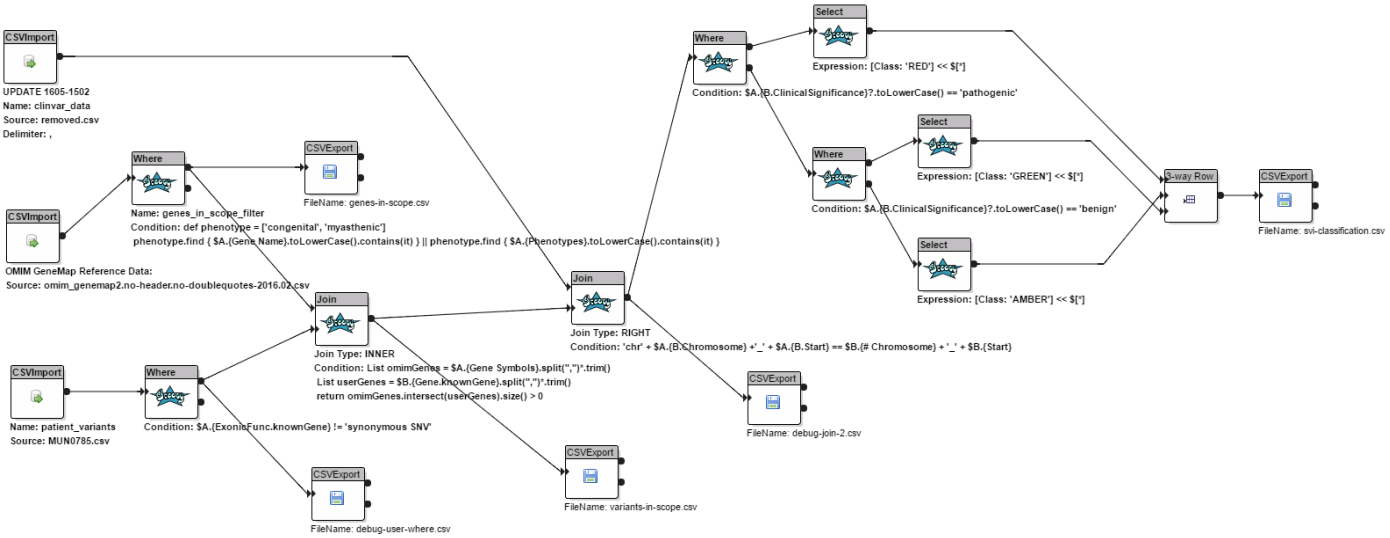

Figure 3: The SVI tool implemented as an e-Science Central workflow.

2.5. Data changes considered in the experiments

As mentioned in the introduction, for each patient SVI uses two kinds of data: the case-specific patient variant file and phenotype, and two external reference databases: OMIM GeneMap to find genes relevant to the given phenotype hypothesis, and NCBI ClinVar used to interpret the pathogenicity of patient variants.

Changes to the reference databases are very relevant because they often affect a large number of patients and are the primary cause of knowledge decay. Furthermore, the SVI reference databases change frequently, providing a good time granularity for experiments: GeneMap updates are published every day, whereas new ClinVar versions are announced every month.

In contrast, changes to patient phenotype are not considered because those represent a change, initiated by a clinician, in the actual disease hypothesis, and this automatically triggers a new investigation for the patient. Similarly, we do not consider updates to the patient variants, because those change infrequently and not enough data points would have been available in our test dataset.

2.6. Experimental setup

For the purpose of this study, SVI was run on a small-scale deployment of the e-Science Central system in Microsoft Azure, consisting of the e-SC server running on a Basic A2 VM (2 CPU-cores, 3.5GB RAM) and of a single workflow engine, hosted on a Basic A3 VM (4 CPU-cores, 7 GB RAM). Both ran Ubuntu 16.04 OS.

We used patient variants files with three phenotypes: Alzheimer's disease, Frontotemporal dementia – Amyotrophic lateral sclerosis and CADASIL syndrome. On average they included 24 thousand records, around 39 MB in size. The OMIM GeneMap reference database was accessed on 31/Oct/2016 (unless stated otherwise) and a range of versions of NCBI ClinVar database were used, from July 2015 to October 2016. For more details about the patient variant files and reference databases please refer to Tab. A.5 and A.6 in Appendix A.

2.7. Baseline: blind re-computation

As mentioned in the introduction, *blind re-computation* refers to the baseline case where any change at all in any of the reference databases triggers a full re-computation of the entire population of prior outcomes. Tab. 1 shows the effects of reacting to new versions of ClinVar regardless of the extent of the changes between any two versions. The Table reports results from nearly 500 executions, concerning a cohort of 33 patients, for a total runtime of about 58.7 hours. As merely 14 relevant output changes were detected, this is about 4.2 hours of computation per change: a steep cost, considering that the actual execution time of SVI takes a little over seven minutes.

Furthermore, the table only portrays a partial picture, as it only includes reactions to monthly changes in Clinvar. A really blind approach would also react to *daily* changes to GeneMap, which are shown to have very little effect on the outcomes. For instance, comparing outputs generated using the same version of ClinVar and four consecutive versions of GeneMap: 16-10-30, 16-10-31, 16-11-01 and 16-11-02 shows that none of the patient variants was affected at all by these small changes.

The sparsity of the table should come as no surprise, as changes in the reference databases are dispersed across the whole human genome and so the chance that they may affect a particular patient are relatively small. Further details on these experiments can be found in supplementary material sheet `CV-blind`.

In the next sections we discuss in detail the three proposed techniques, one for each re-computation problem we have listed in Sec.2.2, namely partial re-execution, differential execution, and scope of change determination.

3. Partial re-execution

We mentioned earlier (Sec. 2.4) that e-SC generates one ProvONE-compliant provenance trace for each workflow run. We exploit these traces to identify the minimal sub-workflow that is affected by the change [7, 8].

Suppose we record a change of the form $d^t \rightarrow d^{t'}$ in reference data, and let I be a past invocation of our workflow. The source blocks for any sub-workflow that is affected by the change are those activities A that were executed as part of I and that used d^t directly. These can be obtained from the query:

:- wasPartOf(A, I), used(A, d^t)

Note that the change event itself can be recorded using a provenance assertion:

wasDerivedFrom$(d^{t'}, d^t)$

In this case, the query becomes:

:- wasDerivedFrom$(d^{t'}, D)$, wasPartOf(A, I), used(A, D)

where D is now a variable that represents the previous version of $d^{t'}$.

Having determined the source blocks, we expand the workflow recursively, by traversing the provenance graph for invocation I, downstream. At each step we seek two possible patterns:

1. execution(A_1), execution(A_2), wasInformedBy(A_2, A_1): given A_1, find all activities A_2 that have been triggered by A_1. This pattern represents a connection from A_1 to A_2, where the intermediate data that flows over the link during execution is implicit;

2. execution(A_1), execution(A_2), wasGeneratedBy(D, A_1), used(A_2, D): Here the dependency between A_1 and A_2 is represented explicitly by the intermediate data product, D.

Fig. 4 shows the sub-workflows related to a change in GeneMap (blue area) and ClinVar (red area). The black arrows on the left indicate the starting blocks for the sub-workflows. The overlapping area between the two sub-workflows contains the blocks that are affected by either of the two changes. Clearly, a partial execution following a change in only one of the databases requires that the intermediate data at the boundary of the blue and red areas be cached.

To provide a measure of the trade-off between additional space requirements and the savings in execution time, we have annotated the figure with the execution time (in seconds) for each of the blocks, and with the size of the cached intermediate data. For example, for a GeneMap change, the corresponding partial execution took about 325 seconds (or 5m:25s), whereas a ClinVar change required 287 seconds (or 4m:47s) to re-execute. Recalling that the cost of a complete execution was 455 seconds (or 7m:35s), these are savings of 28.5% and 37%, respectively. The corresponding additional storage costs are 156MB and just 37KB, resp. The complete results of the partial re-computation following changes in the reference databases are included in supplementary material sheets `CV-subgraph` and `GM-subgraph`.

We note that these figures are obtained from the provenance traces, namely using the standard prov:startTime and prov:endTime properties of activity, along with an additional recomp:dataSize property, which we added to record the size of the entities transferred between blocks.

Fig. 5 provides an alternative view of a possible schedule for the same workflow, which shows the parts of the workflow affected by each of the changes, along with the rendering of the execution times and amount of data involved. We can see that a change in ClinVar affects only a sub-workflow starting in the middle of the workflow, whereas a change in GeneMap affects almost all of the blocks. However, both sub-workflows include the longest running block, which limits the amount of savings that can be achieved.

4. Differential execution

In the previous section we explored options for partial recomputation of a previously executed process. Here we look at a complementary option, namely re-computing P *using only the differences* $diff_D(d^t, d^{t'})$ between two versions of (one or more) reference dataset, D. Some of these ideas are grounded in prior research on the incremental computation and differential computation domains [22, 10].

Using SVI as our testbed again, we show that under some conditions this is feasible to do and results in substantial savings, however, in the general case P requires modifications in order to yield a valid result.

4.1. Computing on data versions differences

To make the idea precise consider again our baseline execution (1):

$$\langle y^t, c^t \rangle = exec(P, x^t, D^t) \tag{4}$$

We are now going to focus on changes to D, thus we assume x^t is constant over time: $x^{t'} = x^t = x$ (in SVI, this means we consider one patient at a time). For simplicity of exposition, initially we also assume a single D with states D^t, $D^{t'}$. In the common case where D is a relation and D^t consists of a set of records, such as a CSV-formatted file (the case for ClinVar), we can express $diff_D(D^t, D^{t'})$ in terms of set differences:

$$diff_D(D^t, D^{t'}) = \langle \delta^+, \delta^- \rangle$$

where:

$$D^{t'} = D^t \setminus \delta^- \cup \delta^+ \tag{5}$$

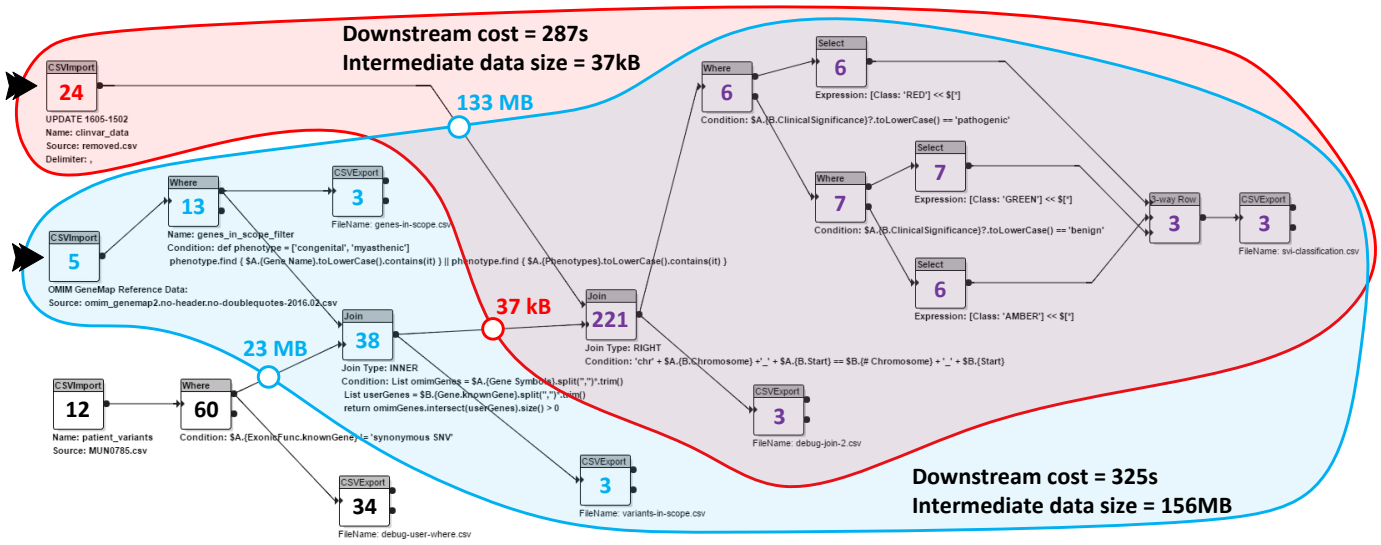

Figure 4: The illustration how retrospective provenance trace of a complete execution can be used to calculate the cost of partial execution; the black arrow at the top-left corner indicates the starting block; the red circle indicates the required intermediate data; red numbers in the workflow blocks denote execution time in seconds.

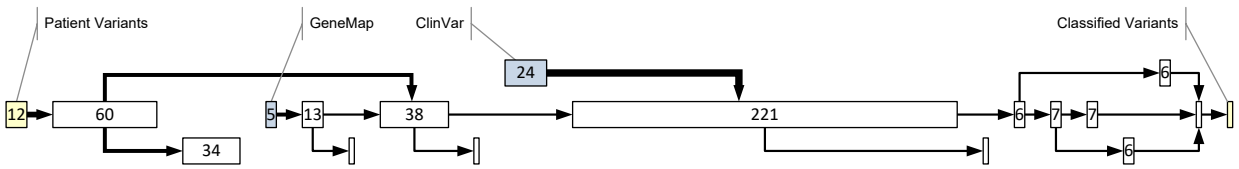

Figure 5: One of the possible schedules of the SVI workflow tasks; wider arrows denote more data flowing between tasks; numbers in boxes represent task execution time in seconds (ClinVar v=16-09, GeneMap v=16-10-31, PV=B_0201).

and δ^+ denotes the added records and *new version of updated* records whilst δ^- are records removed from D^t and the old version of the records that are going to be updated. Note that in the case of ClinVar and most bioinformatics databases, δ^- include *retractions*, which are much less frequent than additions of new records.

Our contention is that the computation of new outcome

$$\langle y^{t'}, c^{t'} \rangle = exec(P, x, D^{t'}) \tag{6}$$

can be broken down into two smaller computations that only use δ^+, δ^- and produce partial outcomes $y^{t'}_+, y^{t'}_-$, which can then be combined with y^t to yield $y^{t'}$. This may require an additional *merge*(.) function that is process-specific. More precisely, we break (6) down into:

$$\langle y^{t'}_+, c^{t'}_+ \rangle = exec(P, x, \delta^+) \tag{7}$$

$$\langle y^{t'}_-, c^{t'}_- \rangle = exec(P, x, \delta^-) \tag{8}$$

$$\langle y^{t'}, c^{t'}_m \rangle = merge(y^t, y^{t'}_+, y^{t'}_-) \tag{9}$$

This breakdown is beneficial if the resulting total cost is less than $c^{t'}$:

$$c^{t'}_+ + c^{t'}_- + c^{t'}_m < c^{t'}$$

Firstly, consider the case where P implements a "well-behaved" function that is distributive over set union and difference. Using (5) and (6) and ignoring cost for the time being, we can write:

$$\begin{aligned} y^{t'} &= exec(P, x, D^{t'}) \\ &= exec(P, x, D^t \setminus \delta^- \cup \delta^+) \\ &= exec(P, x, D^t) \setminus exec(P, x, \delta^-) \cup exec(P, x, \delta^+) \\ &= y^t \setminus y^{t'}_- \cup y^{t'}_+ \end{aligned} \tag{10}$$

Thus, in this case, $y^{t'}_+, y^{t'}_-$ can be automatically combined into $y^{t'}$. However, distributivity is a strong assumption, which does not hold for many practical cases. For example, SVI as a whole distributes only over set difference and union of selected inputs. But in the general case, P may need to be modified in order to combine the partial results using an ad hoc *merge* function.

To illustrate this situation note that SVI essentially consists of four steps (cf. Fig. 2):

a) SELECTION operation that selects from GeneMap only genes relevant to user-defined phenotype ph, producing genes in scope GS:

$$GS = \sigma_{ph}(\text{GeneMap})$$

b) INNER JOIN operation between the input variants, x, and the result of the previous query GS, producing variants in scope VS:

$$VS = x \bowtie GS$$

c) RIGHT OUTER JOIN to combine pathogenicity annotations from ClinVar with corresponding VS yielding VS_p:

$$VS_p = \text{ClinVar} \bowtie\!\!\!\!\subset VS$$

d) final classification into the traffic light system, which adds new classification column to VS_p:

$$classify(VS_p)$$

Given these steps, SVI is specified by the following expression:

$$\begin{aligned} &SVI(x, ph, \text{GeneMap}, \text{ClinVar}) = \\ &\qquad classify(\text{ClinVar} \bowtie\!\!\!\!\subset (x \bowtie \sigma_{ph}(\text{GeneMap}))) \end{aligned}$$

Note, however, that (a), (b) and (d) are all distributive over set union and difference, whereas (c), the right outer join, distributes only for the right-hand side argument. Therefore, we can automatically combine partial outcomes when δ^+ and δ^- are computed for GeneMap, GS, x, VS, or VS_p, whilst differences in ClinVar require a custom merge function.

Regarding GeneMap, the computation steps are as follows. Let

$$y^t = SVI(x, ph, \text{GM}^t, \text{CV})$$

be the original computation, and

$$\text{GM}^{t'} = \text{GM}^t \setminus \delta^- \cup \delta^+$$

be a new version of GeneMap, expressed in terms of version differences. The new $y^{t'}$ can be computed as:

$$\begin{aligned} y^{t'} &= SVI(x, ph, \text{GM}^{t'}, \text{CV}) \\ &= classify(\text{CV} \bowtie\!\!\!\!\subset (x \bowtie \sigma_{ph}(\text{GM}^{t'}))) \\ &= classify(\text{CV} \bowtie\!\!\!\!\subset (x \bowtie \sigma_{ph}(\text{GM}^t \setminus \delta^- \cup \delta^+))) \\ &= classify(\text{CV} \bowtie\!\!\!\!\subset (x \bowtie \sigma_{ph}(\text{GM}^t) \setminus \sigma_{ph}(\delta^-) \\ &\qquad \cup \sigma_{ph}(\delta^+))) \\ &= classify(\text{CV} \bowtie\!\!\!\!\subset ((x \bowtie \sigma_{ph}(\text{GM}^t)) \\ &\qquad \setminus (x \bowtie \sigma_{ph}(\delta^-)) \cup (x \bowtie \sigma_{ph}(\delta^+)))) \\ &= classify(\text{CV} \bowtie\!\!\!\!\subset (x \bowtie \sigma_{ph}(\text{GM}^t))) \\ &\qquad \setminus classify(\text{CV} \bowtie\!\!\!\!\subset (x \bowtie \sigma_{ph}(\delta^-))) \\ &\qquad \cup classify(\text{CV} \bowtie\!\!\!\!\subset (x \bowtie \sigma_{ph}(\delta^+))) \\ &= SVI(x, ph, \text{GM}^t, \text{CV}) \\ &\qquad \setminus SVI(x, ph, \delta^-, \text{CV}) \cup SVI(x, ph, \delta^+, \text{CV}) \\ &= y^t \setminus y^{t'}_- \cup y^{t'}_+ \end{aligned} \tag{11}$$

Although the same approach does not work for changes in ClinVar, we can still adapt SVI and define a bespoke *merge*() function to combine partial results in a way that is *semantically meaningful* for the specific data and process. An implementation of such a function would filter

the results from the right outer join $y_-^{t'}$ and $y_+^{t'}$ by removing rows with null in ClinVar columns, essentially turning right outer join into inner join:

$$z_-^{t'} = filter(y_-^{t'}) \qquad z_+^{t'} = filter(y_+^{t'}) \tag{12}$$

Then, using the filtered products it performs specialised set operations:

$$y^{t'} = y^t \underset{\text{amb}}{\cup} z_-^{t'} \underset{\text{wrt}}{\cup} z_+^{t'} \tag{13}$$

where $a \underset{\text{amb}}{\cup} b$ sets the classification to amber for all rows in a that match b, whereas $a \underset{\text{wrt}}{\cup} b$ overwrites the classification for all rows in a that match b on non-ClinVar columns; both operations leave non-matching rows intact. Given definitions (12) and (13), we can define the specialised merge function as:

$$\begin{aligned} &merge_{SVI}(y^t, y_+^{t'}, y_-^{t'}) = \\ &\qquad y^t \underset{\text{amb}}{\cup} filter(y_-^{t'}) \underset{\text{wrt}}{\cup} filter(y_+^{t'}) \end{aligned} \tag{14}$$

As we can see, this approach may need a substantial amount of process refactoring and makes the definition of *merge* encode some of the process semantics to operate on data differences. Nevertheless, in the rest of this section we are going to illustrate the practical steps in computing the differences δ^+, δ^- and the partial outputs $y_+^{t'}$ and $y_-^{t'}$ on SVI, which will be useful to address the scope analysis.

4.2. Calculating the difference sets

As we have just seen, the reference databases that SVI uses are in the "well-behaved" category of simple relational tables, making it easy to express differences in terms of set operations. Specifically, the added and removed subsets are just set difference between two versions, while the changed subsets are an intersection followed by a selection.

The following SQL-like pseudocode specifies these operations more formally on two versions D_1, D_2 of a data table.

```
ADDED_D1→2   = select * from D2
               where not exists (
                   select 1 from D1
                   where D1.KEY = D2.KEY
               )

REMOVED_D1→2 = select * from D1
               where not exists (
                   select 1 from D2
                   where D1.KEY = D2.KEY
               )

CHANGED_D1→2 = select D2.* from
                   D1 inner join D2 on D1.KEY = D2.KEY
               where D1.NON-KEY_c1 <> D2.NON-KEY_c1 or
                     D1.NON-KEY_c2 <> D2.NON-KEY_c2 or ...
```

These operators assume that we have selected key attributes (possibly compound) for D. Also, $\texttt{CHANGED}_{\texttt{D1}\to\texttt{2}}$ denotes the new version of the updated records, whereas analogous $\texttt{CHANGED}_{\texttt{D2}\to\texttt{1}}$ is used to compute the old version of the updated records. In effect:

$$\begin{aligned} \delta^+ &= \texttt{ADDED}_{\texttt{D1}\to\texttt{2}} \cup \texttt{CHANGED}_{\texttt{D1}\to\texttt{2}} \\ \delta^- &= \texttt{REMOVED}_{\texttt{D1}\to\texttt{2}} \cup \texttt{CHANGED}_{\texttt{D2}\to\texttt{1}} \end{aligned}$$

Note that the CHANGED operator captures all changes in any of the non-key attributes for each record. While this is generic, it ignores the meaning of the attributes relative to the specific processing and is likely to result in a large number of changes irrelevant to the process. For example, two GeneMap records that differ only in the `Comments` attribute would be flagged as different, although the comments are not used anywhere in SVI. Similarly, the only changes in ClinVar records that are relevant to SVI are those in the `ClinicalSignificance` attribute, which drive the classification of variants in the SVI output.

Thus, with the knowledge of the specific use of a relational dataset that the process makes, we partition the attributes into the KEY, USED, and UNUSED datasets. The CHANGED operator can then be rewritten as:

```
CHANGED_D1→2 = select D2.* from
                   D1 inner join D2 on D1.KEY = D2.KEY
               where D1.A1 <> D2.A1 or
                     D1.A2 <> D2.A2 or ...
```

where A1, A2 are attributes from the USED set.

There is an obvious benefit in efficiency resulting from this more aggressive filtering of the difference sets, as illustrated in Tables 2 and 3. The tables report on the number of records of the complete GeneMap and ClinVar datasets and the difference sets calculated using the generic and SVI-specific diff operators. Using the SVI-specific operators the reduction in size is almost always about 90% or over. The only exceptions are the differences between version Jul→Aug 2015 of ClinVar which faced a significant change at the time. Then, the SVI-specific operator yielded a reduction of 49.6%.

More limited gain is achieved when using the generic *diff* operators. In three cases the total size of the difference sets was *larger* than the new version of the ClinVar database. Similarly, the differences between GeneMap 16-06-07 and 16-10-30 computed by the generic operators were only 24.7% smaller than the new version of the database. Clearly, in such cases it is more effective to ignore the difference sets and use only the new version of the data.

Another important aspect of calculating the difference sets is the changing set of attributes. For example, the ClinVar attributes have changed three times since February 2015. These changes in the schema disrupt our difference operators because the three sets KEY, USED, UNUSED change, and also they are no longer perfectly aligned

Table 2: The number of records and reduction percentage of the generic and SVI-based difference sets calculated for selected versions of OMIM GeneMap. Highlighted is less favourable size reduction of the sets.

GeneMap versions $D_{old} \rightarrow D_{new}$	$\lvert D_{new}\rvert$	$\lvert ADDED\rvert + 2 \cdot \lvert CHANGED\rvert + \lvert REMOVED\rvert$		Reduction (%)	
		generic δ	SVI-specific δ	$1 - \frac{\lvert\delta_{gen}\rvert}{\lvert D_{new}\rvert}$	$1 - \frac{\lvert\delta_{SVI}\rvert}{\lvert D_{new}\rvert}$
16-04-28 → 16-06-01	15897	27 + 196 + 1 = 224	27 + 142 + 1 = 170	98.6	98.9
16-06-01 → 16-06-02	15897	0 + 8 + 0 = 8	0 + 4 + 0 = 4	99.95	99.97
16-06-02 → 16-06-07	15910	13 + 76 + 0 = 89	13 + 52 + 0 = 65	99.4	99.6
16-06-07 → 16-10-30	16031	128 + 11944 + 7 = 12079	128 + 636 + 7 = 771	**24.7**	95.2
16-10-30 → 16-10-31	16031	0 + 10 + 0 = 10	0 + 8 + 0 = 8	99.94	99.95
16-10-31 → 16-11-01	16031	0 + 42 + 0 = 42	0 + 0 + 0 = 0	99.7	100.0
16-11-01 → 16-11-02	16031	0 + 4 + 0 = 4	0 + 0 + 0 = 0	99.98	100.0
16-11-02 → 16-11-30	16063	34 + 186 + 2 = 222	34 + 138 + 2 = 174	98.6	98.9

across versions. Therefore, in our implementation of ClinVar diff we assumed that we would compare only columns common in both versions and ignore the added and removed columns. Currently, our SVI re-computation supports any version of ClinVar since Feb 2015.

4.3. Re-computation using the difference sets

To see the effect of using the difference sets on runtime we executed SVI with the range of GeneMap and ClinVar difference sets shown in Tab. 2 and 3. Note that in the case of ClinVar differences we did not include the merge function defined earlier in (14). However, doing so would not affect the runtime significantly as the SVI outputs contain only about a dozen rows. Fig. 6 shows the execution times.

Interestingly, the results indicate two very distinct cases. First, using the difference sets to calculate the output yields clear runtime savings for changes in ClinVar. Re-computation time oscillated around 100 seconds with the only exception for the considerable changes between the July and August 2015 versions of the database (cf. Tab. 3). However, using the GeneMap difference sets we observed loss in the execution time in most cases. Even if the changes were minimal (e.g. 16-06-01→16-06-02 and 16-06-02→16-06-07) and the difference sets contained only a few records, re-execution took about 400 seconds for δ^+ and δ^- separately; over 800 seconds altogether. In two cases we could skip re-execution because the difference sets were empty.

That problem with GeneMap differences stems from the fact that this database is nearly two orders of magnitude smaller than ClinVar. Therefore, the majority of runtime is spent in blocks processing ClinVar whilst the smaller GeneMap file does not affect overall execution time that much. We observed some savings only for two blocks: the WHERE and JOIN located at the front of the pipeline. But the remainder of the pipeline used the complete ClinVar database. Conversely, when ClinVar undergoes changes, the data is used by SVI at the tail of the pipeline where the longest running JOIN block is located (cf. Fig. 5). Thus, using the difference sets rather than the complete version of ClinVar, we could lower the runtime of that block significantly and reduce the total execution time of the relevant workflow subgraph.

Overall, even if using difference sets can reduce runtime of a single partial execution to some extent, the savings depend on the structure of the process and may not be enough to compensate for the fact that two partial executions are needed.

5. Identifying the scope of change

We now address the third problem (P3) from Sec.2.2, namely how to identify the scope of a change in reference data D that is used to produce a large population, Y, of outcomes [4]. As mentioned, SVI is once again a good case study for this problem as the same process is executed over a possibly large cohort of patients (thousands). Whilst these executions are all independent of one another, they all depend on the same reference datasets. The *scope* of a change in any of these dependencies D is subset $Y_s \subseteq Y$ of outcomes affected by change $D^t \rightarrow D^{t'}$.

A possible statistical approach to establishing whether $y \in Y_s$ with some confidence is to sample a number of prior y from Y, compute the corresponding y', and use the differences $diff_Y(y, y')$ to try and learn an estimator for the differences on the unobserved new outcomes. This approach, however, is likely to be sensitive to the specific types of data and process involved and may not always yield robust estimators.

5.1. The basic scoping algorithm

Instead, we propose a scope determination algorithm that relies on the coarse-grained provenance associated with past runs to determine which outcomes y have used a version of D. Coarse-grained provenance, however, only

Table 3: The number of records and reduction percentage of the generic and SVI-based difference sets calculated for 16 versions of ClinVar. Highlighted are less favourable size reductions of the sets.

ClinVar versions $D_{old} \to D_{new}$	$\lvert D_{new}\rvert$	$\lvert ADDED\rvert + 2 \cdot \lvert CHANGED\rvert + \lvert REMOVED\rvert$		Reduction (%)	
		generic δ	SVI-specific δ	$1 - \frac{\lvert\delta_{gen}\rvert}{\lvert D_{new}\rvert}$	$1 - \frac{\lvert\delta_{SVI}\rvert}{\lvert D_{new}\rvert}$
15-07 → 15-08	252656	35087 + 425794 + 85987 = 546868	35087 + 6302 + 85987 = 127376	**−116.4**	**49.6**
15-08 → 15-09	259714	7273 + 16952 + 215 = 24440	7273 + 1342 + 215 = 8830	90.6	96.6
15-09 → 15-10	262498	2832 + 11888 + 53 = 14773	2832 + 1174 + 53 = 4059	94.4	98.5
15-10 → 15-11	277902	15550 + 108588 + 146 = 124284	15550 + 4300 + 146 = 19996	55.3	92.8
15-11 → 15-12	279174	1376 + 489530 + 104 = 491010	1376 + 472 + 104 = 1952	**−75.9**	99.3
15-12 → 16-01	280379	1523 + 23740 + 318 = 25581	1523 + 2224 + 318 = 4065	90.9	98.6
16-01 → 16-02	285041	4710 + 26304 + 48 = 31062	4710 + 1490 + 48 = 6248	89.1	97.8
16-02 → 16-03	286684	2477 + 235330 + 453 = 238260	2477 + 2510 + 453 = 5440	**16.9**	98.1
16-03 → 16-04	290432	3855 + 27088 + 107 = 31050	3855 + 1282 + 107 = 5244	89.3	98.2
16-04 → 16-05	290815	858 + 15732 + 475 = 17065	858 + 1158 + 475 = 2491	94.1	99.1
16-05 → 16-06	306503	18004 + 81738 + 2298 = 102040	18004 + 7174 + 2298 = 27476	66.7	91.0
16-06 → 16-07	320469	14496 + 56692 + 530 = 71718	14496 + 6696 + 530 = 21722	77.6	93.2
16-07 → 16-08	326856	6558 + 58238 + 174 = 64970	6558 + 31356 + 174 = 38088	80.1	88.3
16-08 → 16-09	327632	1020 + 18838 + 244 = 20102	1020 + 1104 + 244 = 2368	93.9	99.3
16-09 → 16-10	349074	22758 + 654486 + 630 = 677874	22758 + 13228 + 630 = 36616	**−94.2**	89.5

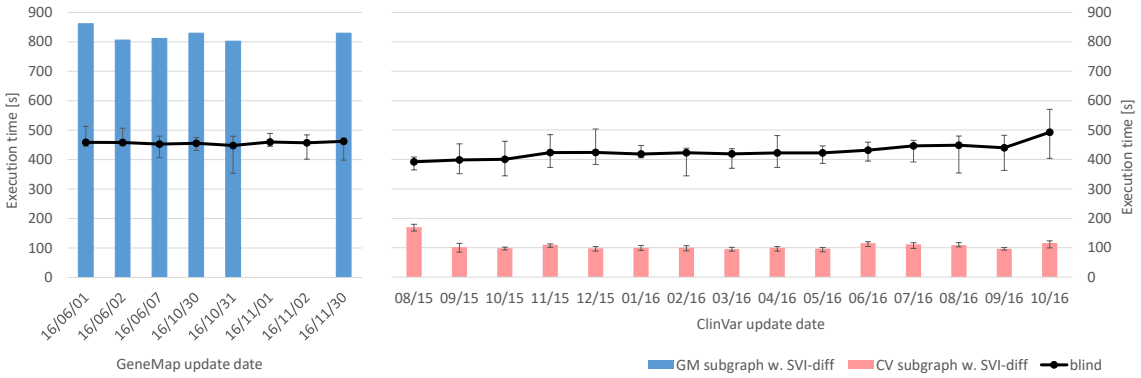

Figure 6: The re-computation time of the SVI workflow using the difference sets following changes in GeneMap (left; ClinVar version 16-08) and ClinVar (right; GeneMap version 16-10-31).

indicates whether or not a dependency on D existed, but not which specific data from version D^t of D was used. It is, therefore, possible that an outcome y that depended on D is not really in the scope of change $D^t \to D^{t'}$, for instance because the process used data from D^t that has not changed in $D^{t'}$. These are *candidate invocations* which must be further analysed to determine the actual impact of change on each of them.

To carry out this further analysis, we propose to re-execute P *one task at a time* using the difference sets. We first consider the case in which the tasks of P are distributive over set union and difference, as described in (10) and (11). Then, we can execute the tasks one at a time until either we observe an empty result or we reach the end of the process. In the former case we know that the invocation was out of scope and no full re-execution is needed. In the latter case we can combine the original output with the final δ^- and δ^+ to obtain the updated result of $P(D^{t'})$. Clearly, this approach is beneficial if computing P on the difference sets is faster than computing P using the entire $D^{t'}$. Algorithm 1 formalises this approach.

Procedure SelectiveExec takes process P and two versions of its input data together with the coarse-grained provenance information represented by *History database* H. First, difference sets δ^+ and δ^- are computed between the two versions of the input data (line 2). Provenance H is then queried in line 3 to find occurrences of statements of form:

$$\mathsf{used}(a, D^t), \mathsf{wasPartOf}(a, I), \ \mathsf{wasAssociatedWith}(I, \_, P)$$

indicating that D^t was used by a specific activity a that

Algorithm 1 Simple selective re-computation of a population of invocations of the process that distribute over set union and difference.

```
1:  procedure SelectiveExec(P, D^t, D^t', H)
2:      ⟨δ+, δ−⟩ = diff_D(D^t, D^t')
3:      I ← ListInvocations(H, P, D^t)
4:      for all I ∈ I do
5:          G ← MinimalSubgraph(I, D^t)
6:          while G ≠ ∅ and (δ+ ∪ δ−) ≠ ∅ do
7:              t ← Pop(G)
8:              δ+ ← t(δ+)
9:              δ− ← t(δ−)
10:         end while
11:         if G = ∅ then
12:             y ← GetOutput(t)
13:             SetOutput(P, D^t') ← y \ δ− ∪ δ+
14:         else
15:             y ← GetOutput(P)
16:             SetOutput(P, D^t') ← y
17:         end if
18:     end for
19: end procedure
```

was part of execution I of process P, or

$$\mathsf{used}(I, D^t), \mathsf{wasAssociatedWith}(I, \_, P)$$

indicating, more broadly, that d^t was used at some unspecified point during I. In both cases the provenance traces identify the set of candidate invocations.

Each of these candidate invocations is then re-executed using the difference sets (loop in lines 4–14). To do it efficiently the algorithm computes the *minimal subgraph* of I that needs re-computation (line 5), as discussed earlier in Sect. 3; for SVI it is one of the subgraphs shown in Fig. 4. The inner loop in lines 6–10 walks through each task of the minimal downstream graph (in topological order) and re-executes the task using the difference sets until either both partial outputs are empty or all tasks have been visited. Assuming that the tasks are distributive under set union and difference, the latter case allows us to generate output of $P(D^{t'})$ using the previous output and partial outcomes of the last task of P (line 12).

For simplicity of presentation the presented algorithm can only work for a linear graph of tasks. Nonetheless, making it work for an arbitrary directed acyclic graph with multiple entry points for D^t is a straightforward extension.

5.2. Practical realisation of scoping

More importantly, the main limitation of Alg. 1 is that it may only be applied across tasks that distribute over set union and difference. That is a strong assumption which is challenging even for a simple example like SVI. Likewise, it is challenging in the much more complex case of NGS pipelines in which the alignment tools (e.g. `bwa`, `samtools`) need access to the complete human reference genome. They cannot perform sequence realignment if given only a difference set between two versions of the reference genome. Thus, to benefit from this algorithm in practice we extended it to allow more diverse process tasks.

Algorithm 2 presents the inner while loop which includes a set of additional checks to make sure that only tasks which can use difference sets properly are considered. Lines 9–11 handle the case from Alg. 1 – distributive tasks. Then, in lines 12–15, the algorithm tries to use the incrementalised version of task t if one is available. That might handle the non-distributive right outer join in SVI following an approach proposed e.g. by [23]. However, as implementing an incremental version of a task is known to be a difficult problem in general, our algorithm includes one more case which we explore in more detail below.

Lines 16–23 handle all other tasks for which we first obtain an impact function. The impact function cannot compute the actual output of t given a partial input. It can, however, determine whether the partial input is *likely* to affect the output of the task. Briefly, the *impT* function returns `true` to denote that the partial input has impact on the output, and `false` otherwise. Given that, if the function returns `true`, the algorithm returns the current task back to the front of G and re-executes the subworkflow using the entire past input of the task (lines 19–21). Afterwards the algorithm exits the loop.

Although the proposed algorithm forces us to implement impact function for all tasks that cannot work with partial inputs, in the simplest *default* implementation it always returns `true` to indicate that any change in the input may affect the output. However, for SVI and the problematic right outer join we were able to use *inner join* accurately.

5.3. Scoping effectiveness

Regarding the effectiveness of this algorithm, note that the blind re-computation would run the minimal subgraph with the complete new data for all $I \in \mathcal{I}$. In contrast, the extended version of our algorithm can reduce the amount of work by making the following assumptions. First, the use of difference sets to calculate output (lines 10–11 and 14–15) is much faster than when using the complete input data. Second, the output of these re-executions is likely to return an empty response, and so the inner while loop can terminate early with $G \neq \varnothing$. Third, the number of non-distributive and non-incremental tasks is small or, alternatively, the provided impact functions are fast, accurate and more effective than the default '`return true`' implementation.

Noting that all these assumptions are valid for SVI, we tested the hypothesis that the approach is indeed beneficial. We show in Tab. 4 that running the process using the proposed algorithm and the SVI-specific diff function we were able to avoid the majority of re-computations which used the complete new ClinVar version. We reduced the number of complete re-executions of the workflow from 495

Algorithm 2 An extension of the selective re-computation over the executions dimension to handle various types of computing tasks. The repeat-until loop in Alg. 1 may be changed as follows.

```
6:  ...
7:  while G ≠ ∅ and (δ+ ∪ δ−) ≠ ∅ do
8:      t ← Pop(G)
9:      if IsDistributive(t) then
10:         δ+ ← t(δ+)
11:         δ− ← t(δ−)
12:     else if IsIncrementalized(t) then
13:         incT ← GetIncremental(t)
14:         δ+ ← incT(δ+)
15:         δ− ← incT(δ−)
16:     else
17:         impT ← GetImpactFunction(t)
18:         if impT(δ+, δ−) = true then
19:             Push(G, t)
20:             tmp_d ← GetInput(t)
21:             ExecuteWorkflow(G, tmp_d)
22:             break
23:         end if
24:     end if
25: end while
26: ...
```

down to 71. In Tab. A.7 in the appendix we show also the re-computation matrix for the algorithm which used the generic diff function. In that case the reduction was less significant and required 302 complete re-executions. That is because the generic diff searched for changes in every single column of the ClinVar data, most of which were irrelevant to SVI.

Figures 7 and 8 show the effect of running our algorithm on the re-computation time. The former presents the average time required to re-compute a single patient variant file. For the majority of cases running SVI with the difference sets was much quicker than with complete ClinVar data. In a few cases, e.g. when using the difference between the versions from September and October 2016, some re-executions were slightly slower than the partial re-computation. That was due to extensive changes in the ClinVar database at the time and so almost all rows were reported as changed. This did not occur, however, when using the SVI-specific diff function. Then, the total time was significantly lower than the partial re-computation in all cases as there were not many changes in the columns relevant to SVI.

Figure 8 shows the total re-computation time for the whole patient cohort including the time required to re-execute tasks with the difference sets and to run the partial re-computation with the complete new data when the impact function produced `true`. This figure emphasises the penalty for running the algorithm when the difference sets were large compared to actual new data. It also highlights the importance of the diff and impact functions. Clearly, the more accurate the functions are the higher runtime

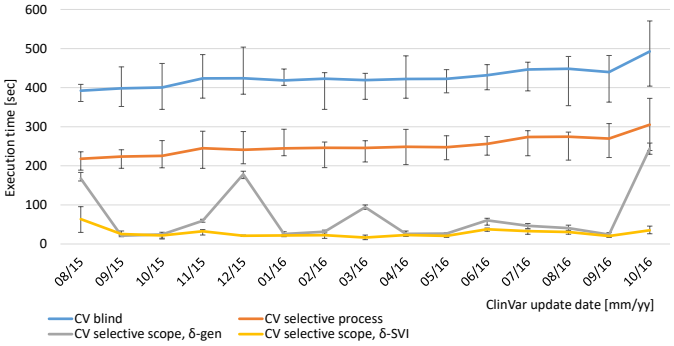

Figure 7: Minimum, average and maximum execution time of the SVI workflow per patient variant file in four approaches to re-computation.

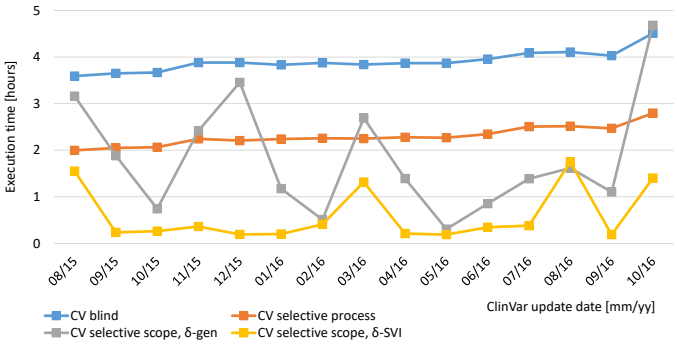

Figure 8: Total time of the SVI workflow for a cohort of 33 patients depending on the approach taken to re-computation.

savings may be, which stems from two facts. Firstly, more accurate diff function tends to produce smaller difference sets which reduces time of task re-execution (cf. CV-diff and CV-SVI-diff lines in Fig. 7). Secondly, more accurate impact function tends to produce `false` more frequently, and so the algorithm can more often avoid re-computation with the complete new version of the data (cf. the number of black squares vs the total number of patients affected by a change in Tab. 4).

6. A blueprint for a generic and automated re-computation framework

So far we have presented techniques that can be applied to reduce the cost of recurring re-computation, with reference to a single case study and without concern for the relative cost and benefits associated with the re-computation.

Our long term goal is to generalise the approach into a reusable framework, which we call `ReComp`, that is able not only to carry out re-computations by automating a combination of the techniques we just illustrated, but also to help decision makers carry out a cost/benefit analysis to determine when selective re-computation is beneficial.

For this, `ReComp` must support a number of capabilities, above and beyond those just illustrated. With reference to our execution model:

$$\langle y^{t'}, c^{t'} \rangle = exec(P, x^{t'}, D^{t'}) \tag{15}$$

Table 4: Changes observed in the output of the SVI tool when executed with the difference sets computed for NCBI ClinVar reference database using the SVI specific δ function; ■ denotes the need for re-execution with the complete new version of ClinVar ($D_{ac} \neq \varnothing$ or $D_r \neq \varnothing$), '·' denotes only task re-execution with the difference sets ($D_{ac} = \varnothing$ and $D_r = \varnothing$).

Phenotype hypothesis	Frontotemporal Dementia-Amyotrophic Lateral Sclerosis												CADASIL	Alzheimer's disease																			
Variant file / ClinVar version	D_1071	D_1049	D_1041	D_0899	D_0854	D_0830	C_0171	C_0098	C_0056	C_0053	C_0051	B_0307	D_1136	C_1457	C_0072	C_0071	C_0068	C_0065	B_0396	B_0384	B_0370	B_0365	B_0358	B_0338	B_0331	B_0229	B_0214	B_0209	B_0208	B_0203	B_0202	B_0201	B_0198
08/15	■	·	·	·	·	·	·	·	·	·	■	·	·	·	■	·	■	■	■	·	■	■	■	■	■	■	■	■	·	■	·	·	■
09/15	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
10/15	·	·	·	·	·	·	·	·	·	·	·	·	■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
11/15	·	·	·	·	·	·	·	·	■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
12/15	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
01/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
02/16	·	·	■	·	■	·	·	·	·	·	■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
03/16	·	·	·	·	·	·	·	·	·	·	·	·	·	■	■	·	■	■	■	·	■	■	■	■	■	■	■	■	■	■	·	■	■
04/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
05/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
06/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
07/16	·	·	·	·	·	·	·	·	■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
08/16	·	·	·	·	·	·	·	·	·	·	·	■	·	■	■	■	■	■	■	·	■	■	■	■	■	■	■	■	■	·	■	■
09/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
10/16	■	■	■	■	■	■	■	■	■	■	■	■	■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·

these are:

1. Detect and quantify changes in input and reference data, i.e. by accepting data-specific $diff_X()$ and $diff_D()$ functions;
2. *Estimate* the impact of those changes on each member $y^t \in Y$ in a population Y of prior outcomes, i.e. learn estimates of $diff_Y(y^t, y^{t'})$ without having to compute $y^{t'}$, as well as estimates of the corresponding re-computation cost $c^{t'}$;
3. Use the estimates to prioritise prior outcomes for re-computation, subject to a limited budget, and
4. Perform the re-computation of the corresponding instances of P, entirely or partially, as we have seen in this paper.

Note that, at this stage, we do not consider changes in P itself or any of its *software* dependencies (as opposed to the data dependencies). For simplicity we focus on changes in the data only and do not consider changes in the underlying processes. These are also relevant but require a separate formalisation, beyond the scope of this paper.

In practice, `ReComp` is configured as a *meta-process* that is able to (i) *monitor* instances of an underlying process P and record its provenance as well as details of its cost, (ii) *detect and quantify changes* in the data used by P, and (iii) *control* the re-execution of instances of P, on demand. These capabilities are summarised in the loop depicted in Fig. 9.

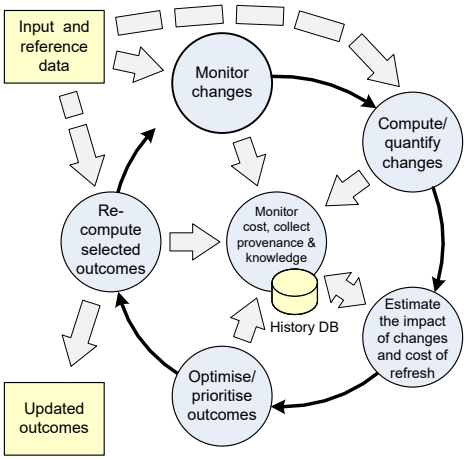

Figure 9: The main loop in the ReComp framework that handles selective re-computation of the user process; thin black arrows denote the flow of control, thick arrows represent the flow of data.

Central to ReComp is the idea that decisions about future re-executions are informed by analytics on the history of past executions. To make this possible, each execution of the form (15) (including re-executions) is controlled by ReComp, and generates metadata records that include:

- outcomes that are subject to revision;
- provenance of the outcome, either coarse-grained or fine-grained, depending on the underlying provenance recording facilities associated with the process runtime;
- execution cost, typically expressed as running time and data storage volume, again as detailed as allowed by the underlying system. For instance, our own WFMS, e-SC, provides block-level time recording and per-data-item storage, while other systems may only provide cumulative times.

Our long-term research hypothesis is that metadata analytics performed on such history database may yield viable models to estimate change impact and thus be able to prioritise re-computations vis-à-vis a limited budget.

In the rest of this section we discuss a number of challenges that underpin the implementation of the ReComp framework.

6.1. Monitoring data changes

Managing multiple versions of large datasets is challenging. Firstly, observing changes in data usually requires source-specific solutions, as each resource is likely to expose a different version release mechanism — a version number being the simplest case. Secondly, the volume of data to be stored, multiplied by all the versions that might be needed for future re-computation, leads to prohibitively large storage requirements. Providers' limitations in the versions they make available also translate into a challenge for ReComp, with some providers not offering access to different versions of their data at all.

A further issue is whether multiple changes to different data sources should be considered together or separately: in some cases it may be beneficial to group multiple changes to one resource instead of reacting immediately. For example, GeneMap updates are published daily, often with only a few rows changed. Thus, taking into account the cost of running the ReComp loop, it may be more effective to delay the loop and collect a number of updates, e.g. over a week.

6.2. Calculating and quantifying changes

Suppose two processes managed by ReComp retrieve different attributes from the same relational database D. Clearly, for each of these processes only changes to the relevant attributes matter. Thus, the *diff* () functions, such as those defined in Sec. 4.2, are not only type-specific but also query-specific. For n processes and m resources, this may potentially require $n \cdot m$ specialised *diff* () functions. Whether we can find more effective ways to compute and measure data changes is an open question. Additionally, some input data may be unstructured or semi-structured and thus calculating the difference between two versions that is useful in the estimation of their impact may be challenging in itself.

6.3. Estimation impact and cost of refresh

We define the re-computation problem as finding the optimal selection of past invocation that can maximise the benefit of re-computation given changes in the input data and a budget constraint. Addressing this problem requires that we first learn impact estimators that can take into account the history of past executions, their cost and the changes in the input data, and can feed into the optimisation problem. This is a hard problem, however, which in particular involves estimating the difference between two outputs of process P given changes to some of its inputs. Clearly, some knowledge of the function that P implements is required, but that is also process-specific and so difficult to generalise into a reusable re-computation framework.

Recalling our example with SVI and ClinVar, we would like to predict whether or not a new variant added to the database will change patient diagnosis. The technique showed earlier allowed us to do so to some extent, as we were able to reduce the number of affected invocations from 495 to 71, yet more work is needed to find more accurate and more generic techniques.

The problem of learning cost estimators has been addressed in the recent past, but mainly for specific scenarios that are relevant to data analytics, namely workflow-based programming on clouds and grid [24, 25]. But for instance [26] showed that runtime, especially in the case of machine learning algorithms, may depend on features that are specific to the input, and thus not easy to learn. That leaves the impact and cost estimation as an open challenge.

6.4. Optimising the selection of past executions

Given a limited re-computation budget, and a measure of benefit of outcome refresh, we can address the further problem to select the past executions that are expected to maximise the benefit given the budget. Using the impact and cost estimators, we can formulate it as the 0-1 knapsack problem in which we want to find vector $\mathbf{a} = [a_1 \dots a_n] \in \{0,1\}^n$ that achieves:

$$\max \sum_{i=1}^{n} v_i a_i \qquad \text{subject to} \qquad \sum_{i=1}^{n} w_i a_i \leq C \tag{16}$$

where n is the number of past executions, v_i is the estimated change impact for execution i, and w_i is the estimated cost of its re-execution. Importantly, each data change event triggers an instance of (16) to be solved but due to expected high cost of re-computation it may be worth grouping a number of change events together. That adds complexity to the optimisation problem.

6.5. Black box processes

Running the SVI example in the previous sections, we assumed that we have insight into the structure and semantics of process P managed by `ReComp`. That enabled us to effectively apply techniques for partial process re-execution. When P is a *black box* process, however, this is not possible and other techniques such as incremental computation [9, 27, 10] may be required. Regardless of the transparency of P, a common challenge is that for big data analytics intermediate data produced by the process (or memoised during incremental computation) often outgrow the actual inputs by orders of magnitude, and thus the cost of persisting all intermediate results may be prohibitive. An open problem, with some contribution from Woodman et al. [12], is to find techniques that could balance the choice of intermediate data to retain in view of a potential future re-computation, with its cost.

A separate challenge is that the actual re-execution of process P used in the past may not be straightforward. It may require redeploying P on a new infrastructure and ensuring that the system and software dependencies are maintained correctly, or that the results obtained using new versions of third party libraries remain valid. Addressing these architectural and reproducibility issues is a research area of growing interest [17, 15, 28, 29].

6.6. History database

As mentioned, `ReComp` needs to collect and store both provenance and cost metadata. Recording cost requires the definition of a new format which, to the best of our knowledge, does not currently exist. Provenance, on the other hand, has been recorded using a number of formats, which are system-specific. Even when the PROV provenance model [20] is adopted, it can be used in different ways despite being designed to encourage interoperability. Our recent study [30] shows that the ProvONE,[9] an extension to PROV, is a step forward to collect interoperable provenance traces, but is still limited as it assumes that the traced processes are similar and implemented as a workflow.

7. Related work

Selective process re-execution has been studied extensively in the past. A distinctive feature of our work, however, is that we propose to look at the problem taking into account three related approaches: partial re-execution, differential re-execution, and scope determination.

Perhaps the best known tool for automated re-execution of programs in reaction to any changes in their dependencies is Make.[10] The tool helps control the build process of a program from the program's source code. Its key feature is the ability to generate a dependency graph between source files, intermediate artifacts and outputs such that a change in one source file results in a partial rather than complete rebuild of program sources. To drive partial rebuild, Make simply uses the file modification date. Whenever any of the prerequisite files has a date newer than the target file, the relevant rule is fired off and the target file is rebuilt.

Two techniques for smart rerun, SRM, and partial process re-execution, in SPADE, are closely related to our work. Smart Rerun Manager (SRM) [7] is part of the Kepler WFMS. The idea of *smart rerun* of a workflow, previously explored by the same group [31], is to react to changes in one or more parameters in a workflow actor by only executing those parts of the workflow that are affected by the changes, taking data dependencies into account. The approach relies on provenance traces and intermediate data collected and stored during workflow execution, and is derived from a similar approach implemented in VisTrails [32]. The idea is that intermediate results from workflow execution can be extracted from a cache instead of recreating them. Each intermediate data product is assigned a unique ID in the cache. The provenance trace is traversed from the end of the execution back to the start. For each actor found during the traversal, SRM checks whether the data products generated by this actor are still valid, i.e. are found in the cache. If that is the case, then the entire subgraph that ends with that actor does not require re-execution. In this case, the cached data is used from that point onwards.

SPADE recently implemented partial process re-execution [8]. The framework can capture fine-grained system-level provenance information and later use it to improve effectiveness of process re-execution. By intercepting the low-level system calls, SPADE can recreate a DAG structure of the process even without explicit workflow specification. The basis of building the acyclic data dependency graph is versioning of the data artifacts. If a task within the process reads and writes to a file, every write generates a new version of the file which can potentially be reused during re-execution and rollback.

Our approach to selective re-computation in the process dimension is similar to both SRM and SPADE, as we collect provenance of workflow-based applications like SRM does. However, to calculate the minimal re-computation subgraph we use the data versioning mechanism provided by e-SC, which is closer to file versioning in SPADE. Also similar is that to store provenance information we use the PROV and ProvONE models, which are successors of the OPM model used by SPADE. Although the idea is not new, ours is the first implementation to operate off the e-SC workflow model, and it is also only a part of a more ambitious picture, where we seek to prioritise re-execution within a large collection of prior outcomes.

Techniques for **incremental computation** address the problem of reacting effectively to incremental changes in the program's input data. An overview can be found in [9, 33]. Briefly, these techniques are based on dependency graphs, memoisation and partial evaluation – con-

[9] https://purl.dataone.org/provone-v1-dev
[10] http://www.gnu.org/software/make

cepts similar to what we use to re-compute our process, yet applied on the scale of a single algorithm or program.

A number of incremental computation solutions also apply to 'big data' problems. Most notable are DryadInc [34], Haloop [35] and Incoop [27]. Again, the main difference between these and our approach is that we consider re-computation in a broader scope, not limited to only a single execution with updated input data. Instead, by combining three approaches we can address the problem of selective re-computation across many executions. The problem we address is how to effectively limit the number of past executions and also the amount of processing a single data update requires. This does not prevent use of other incremental techniques, e.g. differential dataflows [10] or parallel incremental computation implemented in iThreads [36], as the basis for re-execution.

8. Conclusions and future work

Knowledge decay over time is an important issue that affects the value proposition of big data analytics. It is especially important for the next generation sequencing pipelines, in which algorithms and reference data continuously improve. As these pipelines require processing that can easily exceed hundreds of CPU-hours per patient cohort and as they become used on a wider scale,[11] relevant techniques to address knowledge decay and refresh pipeline results are required.

In this paper we presented our investigation into how selective re-computation can help address the knowledge decay issue. Using a case study in the area of clinical interpretation of genetic variants in humans, with a cohort of patients from the Institute of Genetic Medicine (IGM) at Newcastle University, we described three approaches to selective re-computation: at the process level (partial re-execution), data level (differential execution), and whole-cohort level (identification of scope of change).

Regarding partial re-execution, a special role is played by provenance, which we used to build the minimal process subgraph that requires re-execution. For differential execution, we used *diff* () functions to calculate difference sets between two versions of the input data and then, using these sets, to reduce the amount of processing needed. Finally, at the whole-cohort level we showed a significant reduction in the number of patient samples that required refresh. Overall, we were able to lower the cost of re-computation to about 10% of the total time needed for update the previous results. In the immediate future, our plan is to extend the study to a much larger cohort of over 1,500 patients [4], which will provide better figures on actual savings closer to real population scale.

This study informs the more ambitious `ReComp` project.[12] In the long term, we aim to develop a *meta-process* that can observe changes and control re-execution for a variety of underlying, resource-intensive analytics processes, as well as support business-level re-computation decisions vis-à-vis a resource budget. In Sec. 6 we outlined a number of the research and technical challenges associated with this vision.

[11] https://www.genomicsengland.co.uk/the-100000-genomes-project/

[12] http://recomp.org.uk

Acknowledgements

This work is supported by EPSRC grant no. EP/N01426X/1 in the UK and by a grant from the Microsoft Azure Research programme.

References

[1] J. Cała, E. Marei, Y. Xu, K. Takeda, P. Missier, Scalable and efficient whole-exome data processing using workflows on the cloud, Future Generation Computer Systems 65 (2016) 153–168, special Issue on Big Data in the Cloud. doi:10.1016/j.future.2016.01.001.

[2] R. Do, S. Kathiresan, G. R. Abecasis, Exome sequencing and complex disease: practical aspects of rare variant association studies, Human Molecular Genetics 21 (R1) (2012) R1–R9. doi:10.1093/hmg/dds387.

[3] H. Buermans, J. den Dunnen, Next generation sequencing technology: Advances and applications, Biochimica et Biophysica Acta (BBA) - Molecular Basis of Disease 1842 (10) (2014) 1932–1941. doi:10.1016/j.bbadis.2014.06.015.

[4] M. J. Keogh, W. Wei, I. Wilson, J. Coxhead, S. Ryan, S. Rollinson, H. Griffin, M. Kurzawa-Akanbi, M. Santibanez-Koref, K. Talbot, M. Turner, C.-A. Mckenzie, C. Troakes, J. Attems, C. Smith, S. A. Sarraj, C. M. Morris, O. Ansorge, S. Pickering-Brown, J. W. Ironside, P. F. Chinnery, Proof Only, Genome Research 27 (2016) 1–10. doi:10.1101/gr.210609.116.Freely.

[5] P. Missier, E. Wijaya, R. Kirby, M. Keogh, SVI: A Simple Single-Nucleotide Human Variant Interpretation Tool for Clinical Use, in: N. Ashish, J.-L. Ambite (Eds.), Data Integration in the Life Sciences, Springer International Publishing, 2015, pp. 180–194. doi:10.1007/978-3-319-21843-4_14.

[6] E. T. Cirulli, D. B. Goldstein, Uncovering the roles of rare variants in common disease through whole-genome sequencing., Nature reviews. Genetics 11 (6) (2010) 415–425. doi:10.1038/nrg2779.

[7] I. Altintas, O. Barney, E. Jaeger-frank, Provenance Collection Support in the Kepler Scientific Workflow System, Work 4145 (2006) 118–132. doi:10.1007/11890850_14.

[8] H. Lakhani, R. Tahir, A. Aqil, F. Zaffar, D. Tariq, A. Gehani, Optimized Rollback and Re-computation, in: 2013 46th Hawaii International Conference on System Sciences, no. i, IEEE, 2013, pp. 4930–4937. doi:10.1109/HICSS.2013.434.

[9] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, K. Tangwongsan, An experimental analysis of self-adjusting computation, ACM Transactions on Programming Languages and Systems 32 (1) (2009) 1–53. doi:10.1145/1596527.1596530.

[10] F. D. McSherry, D. G. Murray, R. Isaacs, M. Isard, Differential Dataflow, in: 6th Biennial Conference on Innovative Data Systems Research (CIDR '13), 2013.
URL http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[11] Yaxiong Zhao, Jie Wu, Cong Liu, Dache: A data aware caching for big-data applications using the MapReduce framework, Tsinghua Science and Technology 19 (1) (2014) 39–50. doi:10.1109/TST.2014.6733207.

[12] S. Woodman, H. Hiden, P. Watson, Workflow Provenance: An Analysis of Long Term Storage Costs, Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science (2015) 9:1—-9:9doi:10.1145/2822332.2822341.

[13] U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, D. Turkoglu, Traceable data types for self-adjusting computation, ACM SIGPLAN Notices 45 (6) (2010) 483. doi:10.1145/1809028.1806650.

[14] J. F. Pimentel, J. Freire, V. Braganholo, L. Murta, Tracking and analyzing the evolution of provenance from scripts, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 9672, Springer International Publishing, 2016, pp. 16–28. doi:10.1007/978-3-319-40593-3_2.

[15] J. Freire, N. Fuhr, A. Rauber, Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041), Dagstuhl Reports 6 (1) (2016) 108–159. doi:http://dx.doi.org/10.4230/DagRep.6.1.108.

[16] L. C. Burgess, D. Crotty, D. de Roure, J. Gibbons, C. Goble, P. Missier, R. Mortier, T. E. Nichols, R. O'Beirne, Alan Turing Intitute Symposium on Reproducibility for Data-Intensive Research – Final Report.

[17] V. Stodden, F. Leisch, R. D. Peng, Implementing reproducible research, CRC Press, 2014.

[18] H. Hiden, S. Woodman, P. Watson, J. Cała, Developing cloud applications using the e-Science Central platform, Philosophical transactions. Series A, Mathematical, physical, and engineering sciences 371 (1983). doi:10.1098/rsta.2012.0085.

[19] V. Cuevas-Vicenttín, B. Ludäscher, P. Missier, K. Belhajjame, F. Chirigati, Y. Wei, S. Dey, P. Kianmajd, D. Koop, S. Bowers, I. Altintas, C. Jones, M. B. Jones, L. Walker, P. Slaughter, B. Leinfelder, Y. Cao, ProvONE: A PROV Extension Data Model for Scientific Workflow Provenance, Technical report, DataONE Cyberinfrastructure Working Group (may 2016).
URL http://jenkins-1.dataone.org/jenkins/view/DocumentationProjects/job/ProvONE-Documentation-trunk/ws/provenance/ProvONE/v1/provone.html

[20] PROV-DM: The PROV Data Model, Technical report, World Wide Web Consortium (apr 2013).
URL http://eprints.soton.ac.uk/356851/https://www.w3.org/TR/prov-dm/

[21] L. Moreau, P. Missier, J. Cheney, S. Soiland-Reyes, PROV-N: The Provenance Notation, Tech. rep. (2012).
URL http://www.w3.org/TR/prov-n/

[22] Y. Chen, U. A. Acar, K. Tangwongsan, Functional Programming for Dynamic and Large Data with Self-Adjusting Computation, ICFP '14 Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (2014) 227–240doi:10.1145/2628136.2628150.

[23] P.-A. Larson, J. Zhou, Efficient Maintenance of Materialized Outer-Join Views, in: 2007 IEEE 23rd International Conference on Data Engineering, IEEE, 2007, pp. 56–65. doi:10.1109/ICDE.2007.367851.

[24] I. Pietri, G. Juve, E. Deelman, R. Sakellariou, A Performance Model to Estimate Execution Time of Scientific Workflows on the Cloud, in: 2014 9th Workshop on Workflows in Support of Large-Scale Science, IEEE, 2014, pp. 11–19. doi:10.1109/WORKS.2014.12.

[25] M. J. Malik, T. Fahringer, R. Prodan, Execution time prediction for grid infrastructures based on runtime provenance data, in: Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science - WORKS '13, ACM Press, New York, New York, USA, 2013, pp. 48–57. doi:10.1145/2534248.2534253.

[26] T. Miu, P. Missier, Predicting the Execution Time of Workflow Activities Based on Their Input Features, in: I. Taylor, J. Montagnat (Eds.), Procs. WORKS 2012, ACM, Salt Lake City, US, 2012.

[27] P. Bhatotia, A. Wieder, R. Rodrigues, U. a. Acar, R. Pasquin, Incoop: MapReduce for incremental computations, Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11 (2011) 1–14doi:10.1145/2038916.2038923.

[28] L. C. Burgess, D. Crotty, D. de Roure, J. Gibbons, C. Goble, P. Missier, R. Mortier, T. E. Nichols, R. O'Beirne, Alan Turing Institute Symposium on Reproducibility for Data-Intensive Research - Final Reportdoi:10.6084/M9.FIGSHARE.3487382.V2.

[29] R. Qasha, J. Cała, P. Watson, A Framework for Scientific Workflow Reproducibility in the Cloud, in: IEEE e-Science 2016, no. October, 2016.

[30] W. Oliveira, P. Missier, K. Ocaña, D. de Oliveira, V. Braganholo, Analyzing Provenance Across Heterogeneous Provenance Graphs, in: Ipaw, Vol. 5272, 2016, pp. 57–70. arXiv:1406.2495, doi:10.1007/978-3-319-40593-3_5.

[31] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, Y. Zhao, Scientific Workflow Management and the Kepler System, Concurrency and Computation: Practice and Experience 18 (10) (2005) 1039–1065. doi:10.1002/cpe.994.

[32] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, H. T. Vo, Vistrails: Enabling interactive multiple-view visualizations, in: Visualization, 2005. VIS 05. IEEE, IEEE, 2005, pp. 135–142.

[33] G. Ramalingam, T. Reps, A categorized bibliography on incremental computation, Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93 (1993) 502–510doi:10.1145/158511.158710.

[34] L. Popa, M. Budiu, Y. Yu, M. Isard, DryadInc: Reusing work in large-scale computations, ... workshop on Hot Topics in Cloud ... (2009) 2–6.
URL http://static.usenix.org/events/hotcloud09/tech/full{_}papers/popa.pdf

[35] Y. Bu, B. Howe, M. D. Ernst, HaLoop: Efficient Iterative Data Processing on Large Clusters, Proceedings of the VLDB Endowment 3 (1-2) (2010) 285–296. arXiv:arXiv:1209.2191v1, doi:10.14778/1920841.1920881.

[36] P. Bhatotia, P. Fonseca, U. A. Acar, B. B. Brandenburg, R. Rodrigues, iThreads, ACM SIGARCH Computer Architecture News 43 (1) (2015) 645–659. doi:10.1145/2786763.2694371.

Appendix A. Input data

Table A.5: Basic properties of a set of patient variant files used in the experiments.

Phenotype hypothesis	Variant file	Record count	File size [MB]
Alzheimer's disease	B_0198	23,803	38.5
	B_0201	24,809	39.9
	B_0202	24,442	39.4
	B_0203	24,654	39.8
	B_0208	24,264	39.1
	B_0209	24,166	39.1
	B_0214	23,370	37.9
	B_0229	24,133	39.0
	B_0331	23,897	38.8
	B_0338	24,243	39.2
	B_0358	24,181	39.1
	B_0365	24,070	38.9
	B_0370	23,798	38.4
	B_0384	24,905	40.2
	B_0396	23,886	38.8
	C_0065	23,469	38.0
	C_0068	24,098	39.0
	C_0071	23,741	38.4
	C_0072	22,946	37.3
	C_1457	23,649	38.3
CADASIL	D_1136	24,511	39.6
Frontotemporal dementia – Amyotrophic lateral sclerosis	B_0307	24,052	39.0
	C_0051	23,921	38.7
	C_0053	23,980	38.8
	C_0056	23,805	38.6
	C_0098	22,948	37.4
	C_0171	24,387	39.6
	D_0830	24,132	39.1
	D_0854	24,133	39.0
	D_0899	24,034	38.8
	D_1041	24,463	39.5
	D_1049	24,473	39.5
	D_1071	24,102	39.0

Table A.6: Basic properties of the OMIM GeneMap and ClinVar reference databases used in the experiments.

Database	Version	Record count	File size [MB]
OMIM GeneMap	16-04-28	15,871	2.65
	16-06-01	15,897	2.66
	16-06-02	15,897	2.66
	16-06-07	15,910	2.66
	16-10-30	16,031	2.69
	16-10-31	16,031	2.69
	16-11-01	16,031	2.69
	16-11-02	16,031	2.69
	16-11-30	16,063	2.70
NCBI ClinVar	15-07	304,207	95.0
	15-08	252,656	81.6
	15-09	259,714	87.1
	15-10	262,498	88.1
	15-11	277,902	93.5
	15-12	279,174	94.5
	16-01	280,379	94.8
	16-02	285,041	96.6
	16-03	286,684	94.7
	16-04	290,432	96.1
	16-05	290,815	96.1
	16-06	306,503	101.4
	16-07	320,469	106.7
	16-08	326,856	109.2
	16-09	327,632	109.5
	16-10	349,074	121.3

Table A.7: Changes observed in the output of the SVI tool when executed with the difference sets computed for NCBI ClinVar reference database using the generic δ function; ■ denotes the need for re-execution with the complete new version of ClinVar ($D_{ac} \neq \varnothing$ or $D_r \neq \varnothing$), '·' denotes only task re-execution with the difference sets ($D_{ac} = \varnothing$ and $D_r = \varnothing$).

Phenotype hypothesis	Frontotemporal Dementia-Amyotrophic Lateral Sclerosis												CADASIL	Alzheimer's disease																			
Variant file / ClinVar version	D_1071	D_1049	D_1041	D_0899	D_0854	D_0830	C_0171	C_0098	C_0056	C_0053	C_0051	B_0307	D_1136	C_1457	C_0072	C_0071	C_0068	C_0065	B_0396	B_0384	B_0370	B_0365	B_0358	B_0338	B_0331	B_0229	B_0214	B_0209	B_0208	B_0203	B_0202	B_0201	B_0198
08/15	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
09/15	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
10/15	·	·	·	·	·	·	·	·	·	·	·	·	■	■	·	·	·	■	·	·	·	■	·	·	■	■	·	■	■	·	·	■	■
11/15	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
12/15	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
01/16	·	·	·	·	·	·	·	·	·	·	·	·	·	■	■	·	■	■	■	·	■	■	■	■	■	■	■	■	■	■	·	■	■
02/16	·	·	■	·	■	·	·	·	■	·	■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
03/16	■	·	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
04/16	·	·	·	·	·	·	·	·	·	·	·	·	·	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
05/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	■	·
06/16	·	·	■	·	■	·	■	■	·	·	■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
07/16	·	·	■	·	■	·	·	·	■	·	■	·	·	·	■	·	·	·	·	■	■	·	·	■	·	■	■	·	■	■	■	■	■
08/16	·	·	·	·	·	·	·	·	·	·	·	■	·	■	■	■	■	■	■	·	■	■	■	■	■	■	■	■	■	■	·	■	■
09/16	·	·	·	·	·	·	·	·	·	·	·	·	·	·	■	·	■	■	■	·	■	■	■	■	■	■	■	■	·	■	·	·	■
10/16	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■